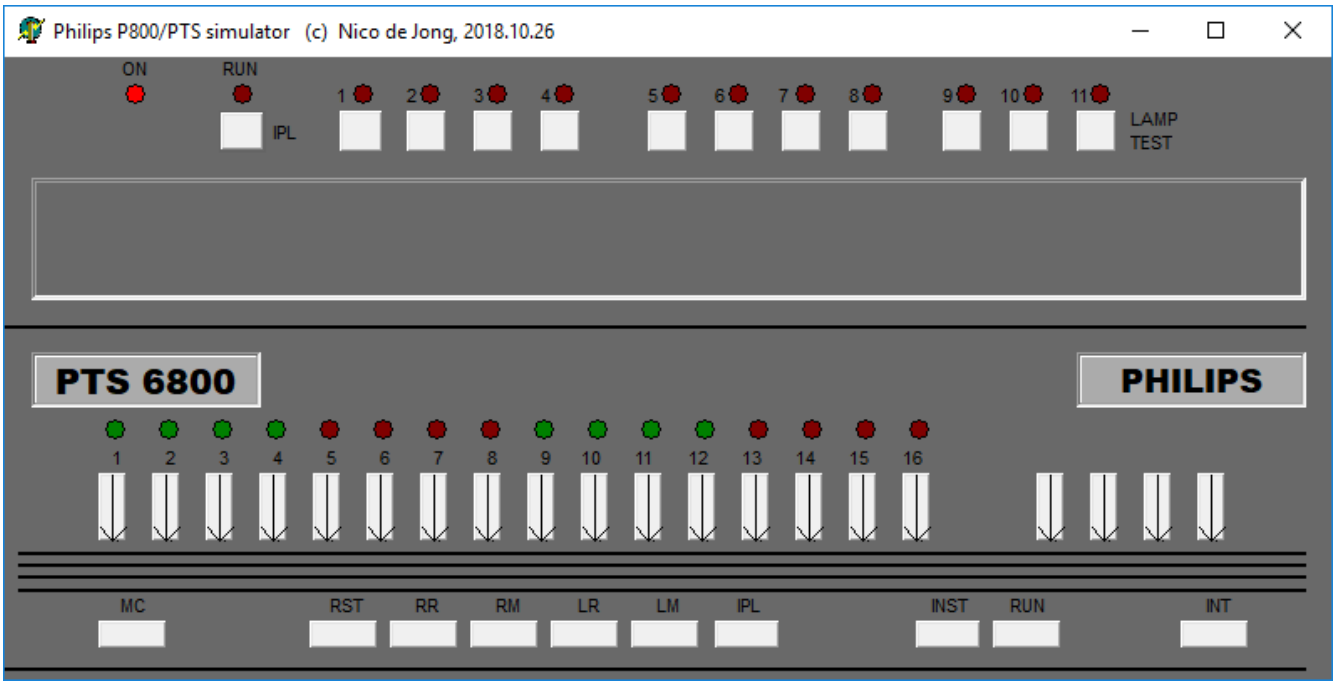


PHILIPS P800 SIMULATOR HANDBOOK

Preface

System under construction, november 2018

The reason for this manual being made, was the fact that the danish IT museum, www.datamuseum.dk, back in the 1980's, inherited a complete Philips PTS development system from Farum Data ApS, which again had received the system from Philips Data Systems A/S, in order to be able to continue support to the few PTS customers left.

Shortly before I retired, I got the time to do something about the remaining PTS parts (some critial parts had disapperared), and I decided to take the remaining parts home, and see if could get a working system out of it.

At present, what was used as the development system, is working, although without the (missing) harddrives.

There were sufficient spare parts left to convert the expansion box (used when you had too many adapters in the main system) into a small system. However, this system could not support any external units, as there was nothing useful left.

The idea therefore arose to write a simulator, so we could enter small programs into that 'tabletop' system, and this manual is one of the results of this project.

This manual describes the Assembler and the Simulator

Part 1 : the Assembler

Introduction

As usual, an Assembler analyses a programs source text, and produces a file containing information, which, when loaded into the computer, will execute the program.

Below, a small part of a source text and the accompanying output listing is given. The object code, which is loaded into the computer, is not shown as such, as it is not printable, but you can see it in the output listing.

Source text :

```
                AORG    /0080                   SPACE
FLOP0           EQU     /04                     DEVICE ADDRESS.
                DATA    /FFFF,/0000
*
* FILL THE SEGMENT BUFFER WITH 4 X 32 CHARACTERS
START           LDK     A1,'A'                  LOAD 'A' INTO REG 1
                LDK     A2,32                   LOAD SPACE INTO REG 2
                LDK     A3,4                    LOAD 4 INTO REG 3
                LDKL    A7,SEGMENT              LOAD BUFFER ADDRESS
WRITE           LDR     A8,A1                   COPY A1 TO A8
                ANKL    A8,/00FF                ERASE LEFT PART OF A8
                SLA     A1,8                    MOVE CONTENTS OF A1 TO LEFT HALF
```

Output listing :

```
                                *
0014                            AORG    /0080       SPACE
0015                            FLOP0   EQU     /04         DEVICE ADDRESS.
0016  0080  FFFF  0000           DATA    /FFFF,/0000
                                *
                                * FILL THE SEGMENT BUFFER WITH 4 X 32 CHARACTERS
0019  0084  0141                START   LDK     A1,'A'      LOAD 'A' INTO REG 1
0020  0086  0220                        LDK     A2,32       LOAD SPACE INTO REG 2
0021  0088  0304                        LDK     A3,4        LOAD 4 INTO REG 3
0022  008A  8720  00E2                   LDKL    A7,SEGMENT  LOAD BUFFER ADDRESS
0023  008E  8084                WRITE   LDR     A8,A1       COPY A1 TO A8
0024  0090  A0A0  00FF                   ANKL    A8,/00FF    ERASE LEFT PART OF A8
```

The leftmost column is just a line counter.

The second column (0080, 0084....) represents the location counter (in Hexadecimal), and the third and fourth column represent the object code.

You can see that the object value /0141 is the machine-readable version of loading a capital 'A' into register 1, as generated by the Assembler.

/01

The lines with AORG /0080 and DATA /0000,/FFFF should always be present.

The first /0080 (= 128 decimal) bytes were used for loading a boot program, residing in a (P)ROM, or, in some versions, in a diode matrix.

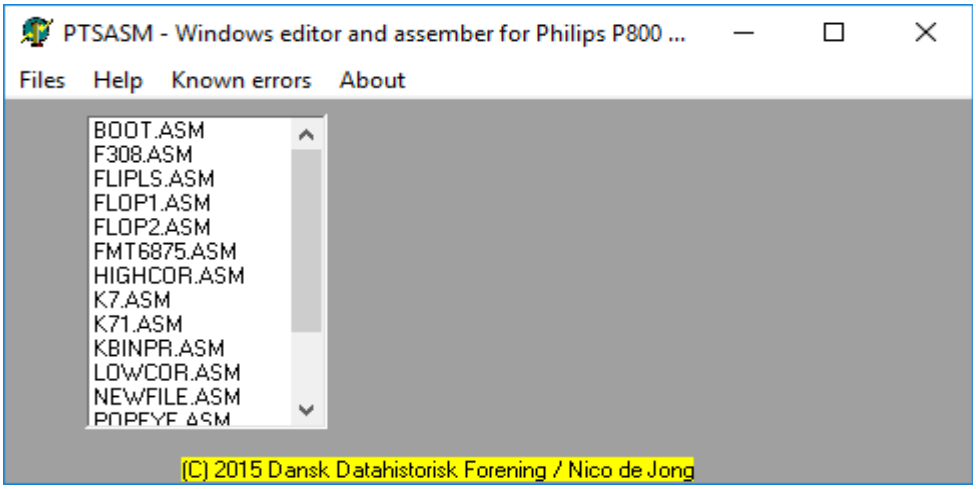
So, when you pressed the IPL button, a small program was loaded into the first 128 bytes, which then presented a possibility to select a program (e.g. the program you just compiled) from a cassette, floppy disc or from the harddisc.

It is beyond the scope of this manual to teach you how to be an Assembler programmer, but programmers knowing about one assembler can quickly pick up how to do it the Philips way, as the Simulator goes into some detail on how it works.

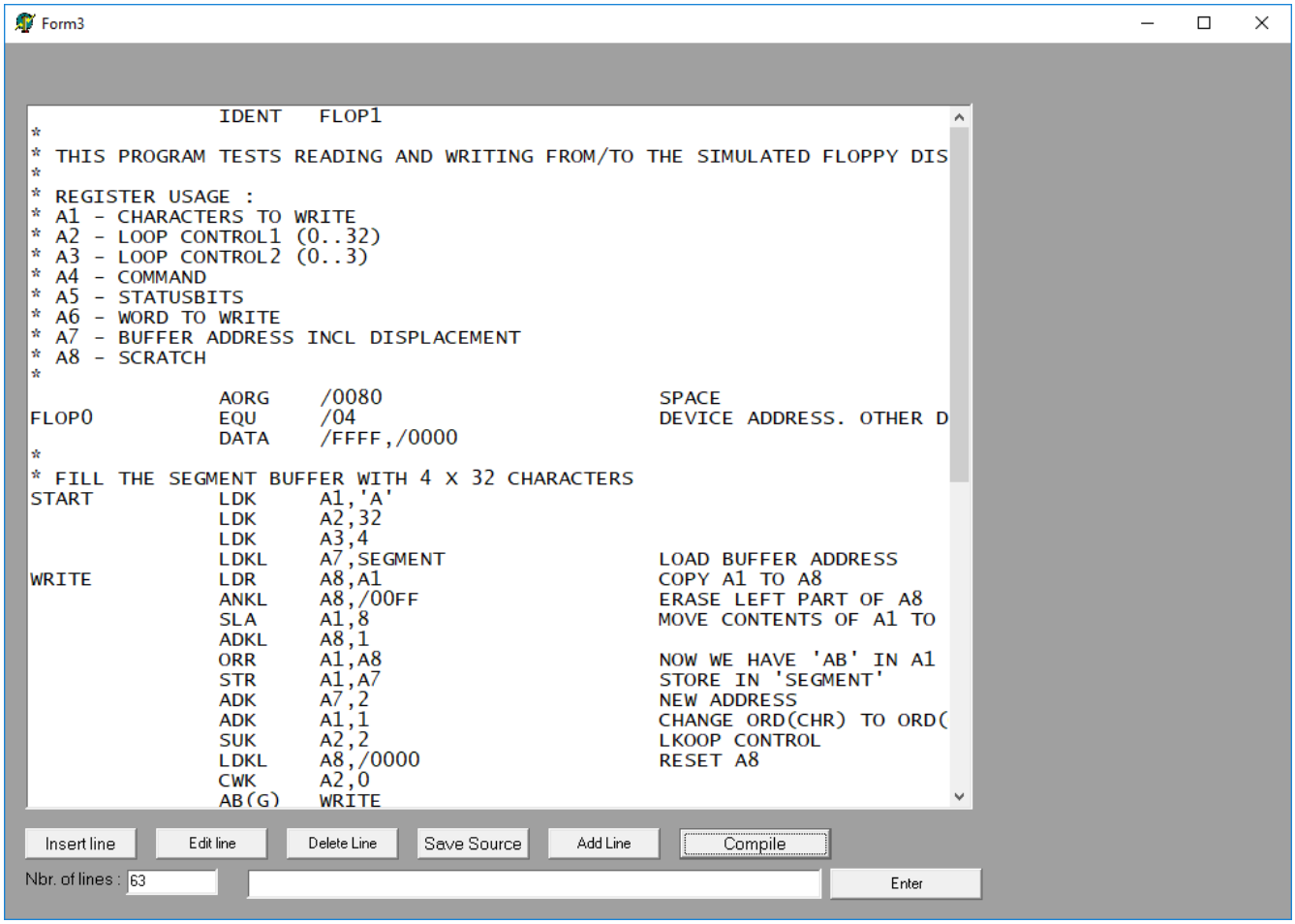
There is not much more to tell about the assembler, but I would be grateful for any comments you might have.

Starting the Assembler

Having started the program, and having pressed 'Files', the screen below is presented :



Selecting FLOP1 as an example, presents the following screen :

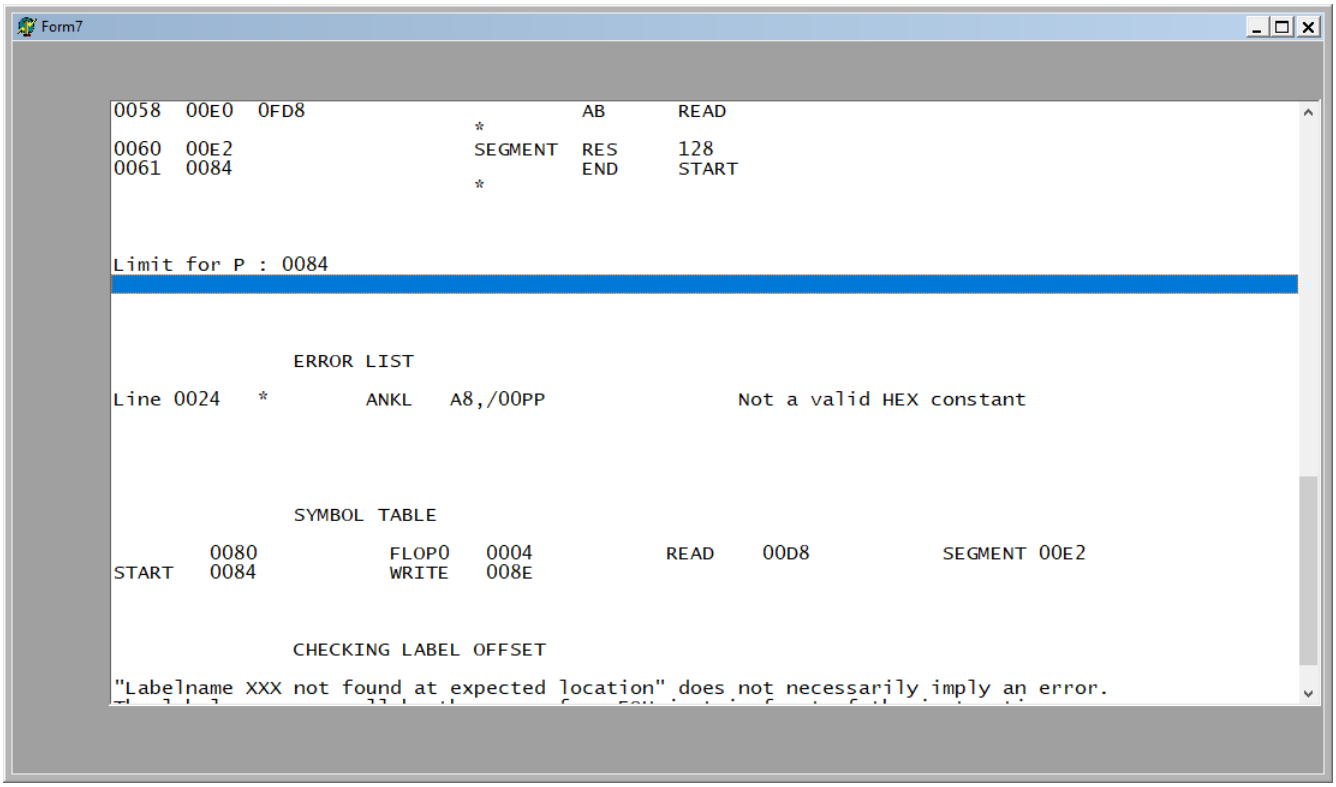


This is the source text of the program FLOP1.

Now you can edit the source, but it is not recommended to do all of your text editing this way. The editor is not very effective, but it is very useful to make minor changes, the main benefit being that you can change the source and compile it (and thus finding errors), before saving your changes to the harddisc ('Save Source').

'Compile' analyses the contents of the listbox and attempts to assemble it, meaning that it tries to find logical errors.

Now suppose that you changed constant in ANKL A8,/00FF to /00pp, which is not a valid hexadecimal value. This would generate an error, like shown below :



Below the error list, you can see the Symbol Table, which is a table showing the location of all labels and EQUates.

The assembler is probably not error free, so it is always a good idea to check whether the addresses in the Symbol Table correspond to the addresses given in the printer listing. If not, you could have written an invalid instruction. Remember, this assembler is not a commercial product.

The best way to start, is to compare the value of 'Limit for P' with the line just above the END statement.

If you find that these addresses do no match, check the other symbols, going up in the list.

When you scroll back to the source listing, you will see that the code generated for the faulty instruction is /A0A0 /0000. This is because the constant is invalid, so the assembler replaces it with zeroes.

This is very useful when you start debugging your program, as you can manually replace the zeroes by the value you need. How to do that, will be explained in the Simulator part.

I

Part 2 : the Simulator

First of all : what IS a Simulator? As the name says : it simulates something.

In this case, a program simulating the workings of a Philips P800 minicomputer, or to be more precise : being able to read and execute programs, designed for and working on a P800 minicomputer, on a PC, without change.

This is of course a truth with modifications, as for example input/outroutines talking to devices that are not available on a PC, e.g. an 8” 255K floppy drive, are to be emulated.

So, in the context of this manual, I define 'simulation' as being able to interprete and execute an instruction, while 'emulation' is the execution of the input/output, corresponding to a PC driver.

In a production environment, a simulator is mostly invisible. An example of this is the IBM 360/370 series, which had a simulator for the IBM 1401. This enabled the execution of 1401 programs on 360/370, which made it unnecessary to rewrite all 1401 programs when systems were changed. However, in some/most cases minor changes had to be done anyway. Apart from that, 1401 programs were written in a language called Autocoder, which was not available on the 360/370.

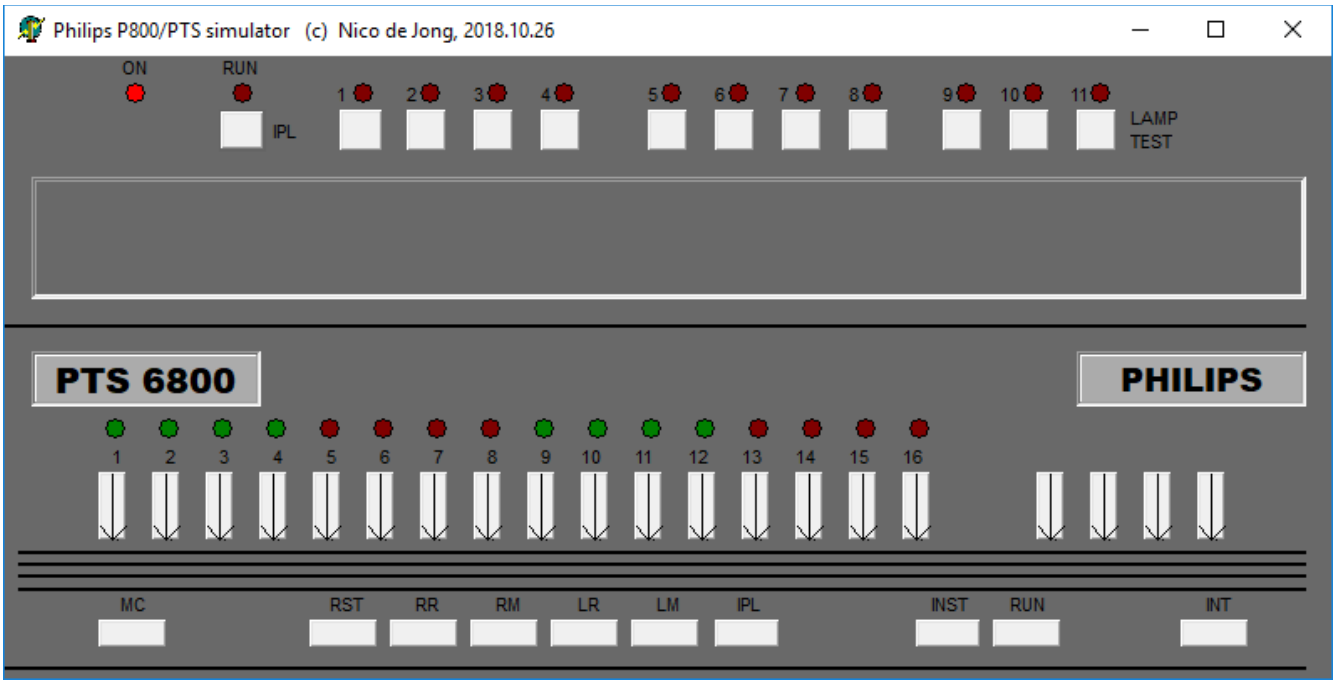
As this project was (also) meant for educational purposes, it was beneficial to build a debugger 'on top' of the simulator.

A debugger is a function where you can look at variables, change things, and do many other things while the program is running, all in order to be able to detect errors, make temporary patches, repeat portions of the code, etc. etc.

Debuggers are especially worthwhile when using assemblers, as it is extremely easy to goof in this kind of machine-near (I would not say 'low-level') program-ming.

Even if you are not a P800 professional, you will find many nice things, which you might be able to use in your own system.

So, enjoy the simulator.



First, let's have a look at the panels. They are a graphical representation of the panels as found on my 'table top' system.

The top panel is called a SOP (Systems Operator Panel). This panel is always present, as it enables the operator to reboot the system by pressing IPL. Depending on the contents of the IPL PROM, a number of LED's (1..11) are lit up, enabling the selection of a program from a floppy disc or the cassette drive.

The LAMP TEST function can be used to ensure that all LED's are working.

The bottom panel is called the CFP (Computer Full Panel) which is used when you need to do more then just starting a program.

We see 4 x 4 lights, which can be lit and/or extinguished from the program and/or the operator.

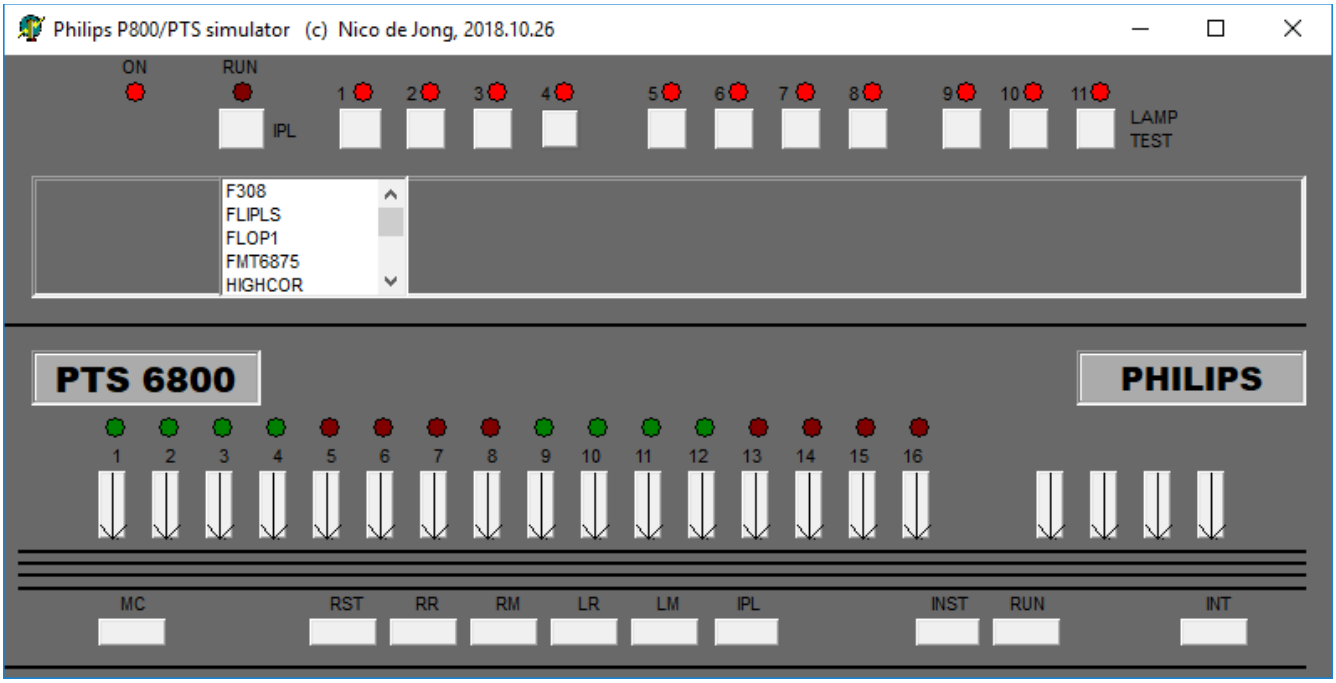
Below the lights, we see 16 tumble switches, all in the LOW (Off) position. These are used to enter data or addresses into the system. To the right of the 16, we see 4 switches, which are used to specifiy which register you want to read or modify.

The bottom row of push buttons have the following meaning :

- MC Master Clear Clears memory, interrups, etc.
- RST Read Status Word Displays SW on the 16 lamps
- RR Read Register as specified by the 4 switches
- RM Read memory as specified by the 16 switches

- LR Load Register as specified by the 4 switches
- LM Load Memory
- IPL Initial Program Load Load the lowest 128 bytes with a boot prg
- INST Execute 1 instruction
- RUN Run the program
- INT Interrupt a running program

How to use the use these buttons, is explained in Appendix A.

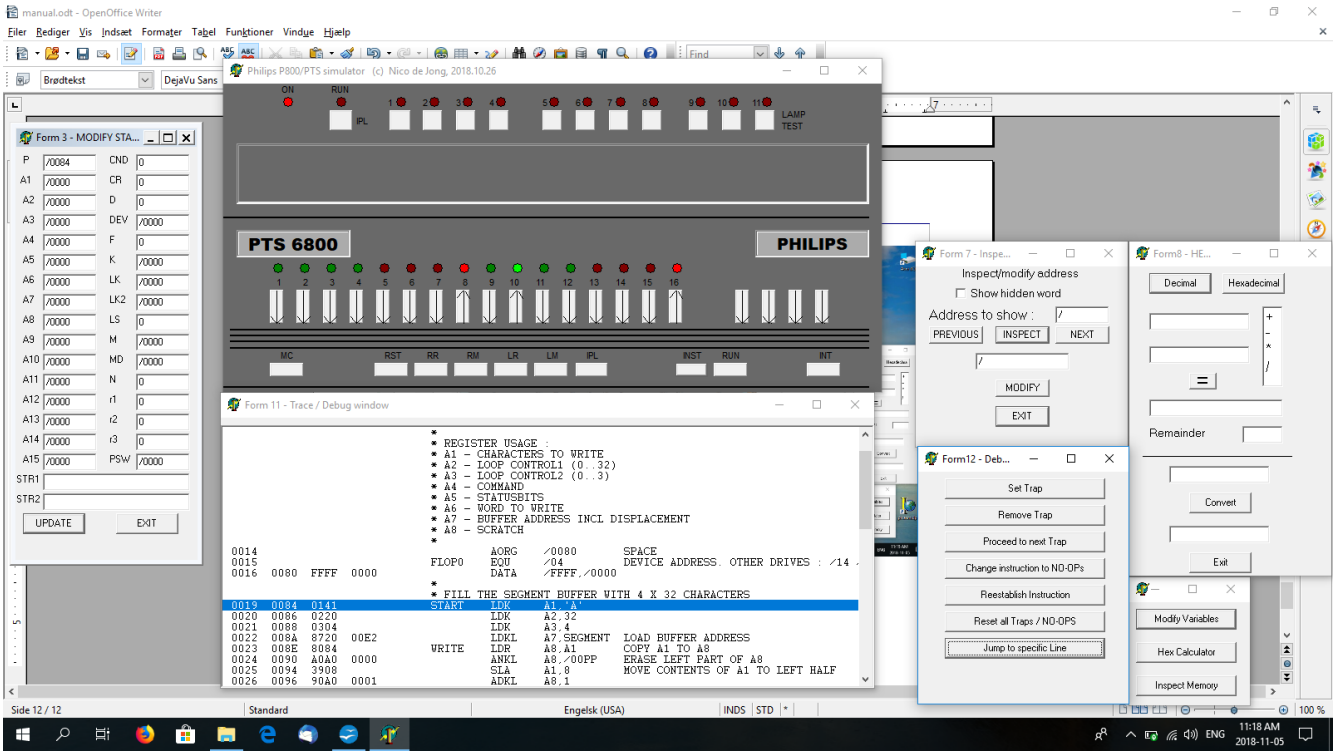


Selecting and starting a program is very easy.

Press **IPL.** You now have two possibilities. If you have only a few programs, you can put the cursor on one of the SOPs 11 pushbuttons, and see which program will be started when you press that button.

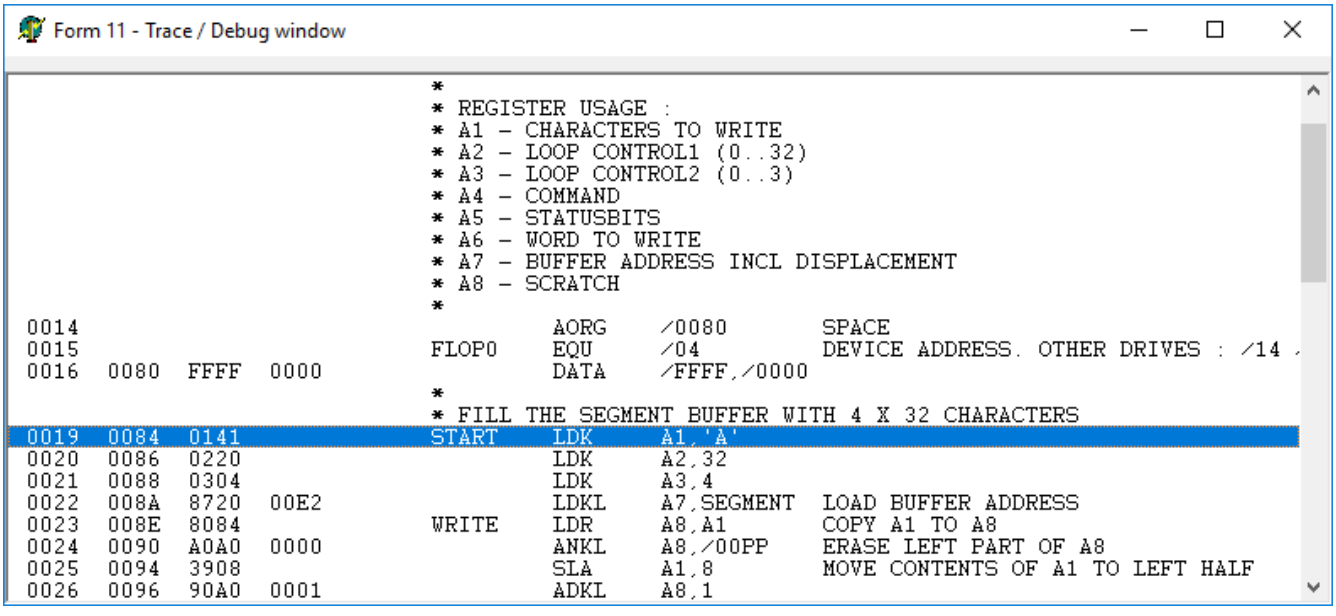
An alternative, but not quite Philips style, is to press the **SOP's IPL** button. This will present you for a listbox containing the names of all *.ASM files in the directory. Select the relevant program.

Now you can start the Simulator by pressing **INST** which will present an utterly confusing screen, shown on the next page.



Confused ? You will not be a few pages down the line.

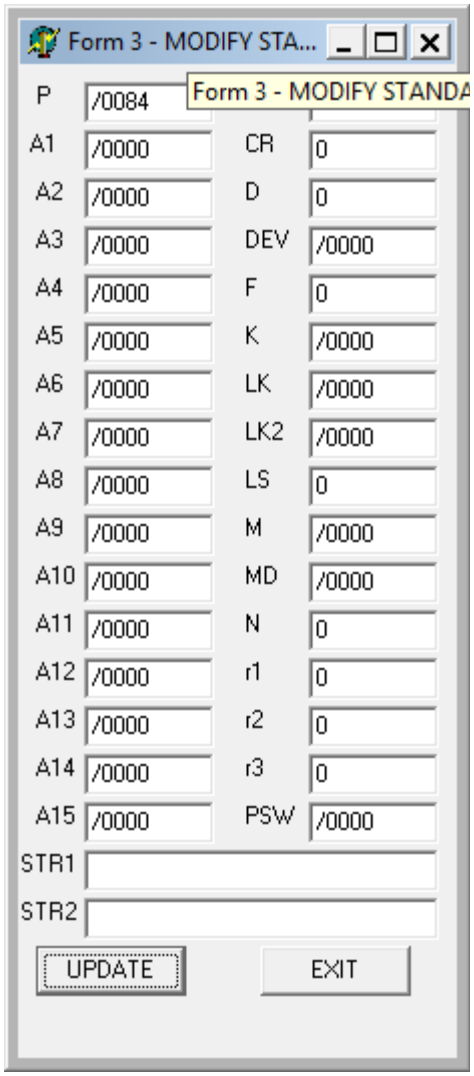
We start with the panels we saw before. The lights now show a value /0141.
Now look at the 'Trace Debug' window :



You will note that the blue line on line 19, also “covers' the object code /0141. So, the object code to be executed is shown in the lights on the panel.

But there is more to it.

Look at the 'Modify Standard Variables' frame.



You can see, that the P register, in fact the location counter, shows /0084, which is the location where the instruction is stored. The rest of the fields is still empty, as we have not done anything yet.

When you now depress **INST** the contents of the A1 field will be changed to /0041 which is the hexadecimal value of an A.

And that is what the debugger is all about : it shows you what happens when you press a button, and, BTW, the lamps on the FCP have changed too.

Most other fields in the Modify Standard Variables screen can have been changed too, so this is the place to explain why that is so.

The P800 is a mini computer with core memory (core as in magnetic), which is very expensive, so the size of the memory was to be kept low. The effect of this was, that most instructions are only 2 bytes long, which apart from being cost effective also meant that they could be executed quickly. The drawback was that e.g. small constants had to be a part of the instruction.

Let me give an example

/0141 tells us to put an A into register 1.

In a 'normal' PC, this would take three words : one for the load command, one for the specification of register 1, and one for the constant.

The specification for the LDK command is in binary **0000 0**rrr kkkk kkkk

where rrr are 3 bits specifying a register 0..7, and kkkk kkkk 8 bits specifying the constant. And what does this also imply ? You cannot load a short constant (8 bits) into register 8..15 !

But what do you have to do to load a short or a long constant into e.g. register 8?

Well, there is an instruction LDKL (Load constant long), which has the bit pattern

1000 0rrr r**010 0000**

The first 3 r's define again the registers 0..7. the 4th r tells us 'add 8 to it', so if we take rrrr as a whole, the binary values of the bits can be interpreted from the left as 4218.

The constant itself is located in the word following the **LDKL** instruction.

All these specialities are explained in the P800 Assembly Manual.

The various bits can have different meanings for different instructions. Another example : look at the fields **LK** and **LK2**. They contain the next 2 instructions, and why is that ?

Well, if we treat the present instruction as if it was a LDKL, the word behind the instruction would be a constant. The same is valid for LK2, as there are a few instructions needing three words.

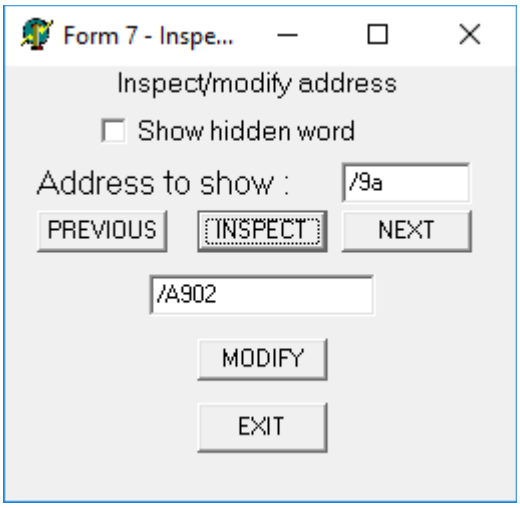
In other words, when I decode an instruction, I extract all variables, regardless of which instruction is to be decoded.

This frame is very useful, as you can change all fields. You could e.g. change the instruction counter (**P**) so you could jump over a specific instruction, or you could execute the same instruction again, possibly with a different value in the Condition Register (useful for testing Branch instructions).

Dont forget that changing one thing, e.g. the field **A1** also will change the field **D**, as this is the same value, but in decimal !

Now that you master this part of the Simulator/Debugger, we can look at the remaining three parts :

Inspect/modify address



This frame is very useful if you want to inspect for example a part of a string or an element in an array. Just enter the address in the 'Address to show' field, preceeded by a slash.

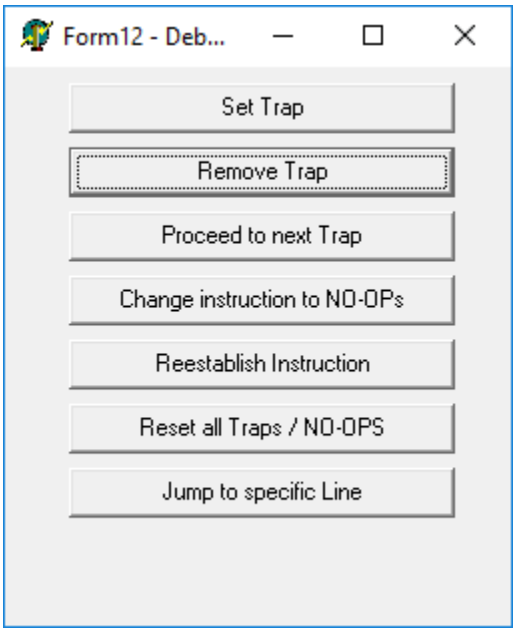
Press **Inspect** and the contents of that word will be displayed. You can also scroll through addresses in the vincinity of the word specified, by pressing **Previous** or **Next**. You can also modify the words by changing the value and then pressing **MODIFY**.

But what is 'Show hidden word' ?

Here we have a strange situation, caused by the fact that when we look at the printed list, we see the addresses expressed as bytes (0080, 0082 etc). However, when using a PC, we use Words.

This is a bit awkward to handle, so the simulator saves 2 bytes (in fact a whole instruction) in one word, but addresses words only. This means, that we waste half of the space, but this has a nice side effect, which will be explained in detail when we get to trap-setting. Let it be sufficient right now that we can see the contents of the 'wasted' words.

Debug aids



This frame enables you, to put it bluntly, to screw things up, but it is also a valuable tool for testing a program.

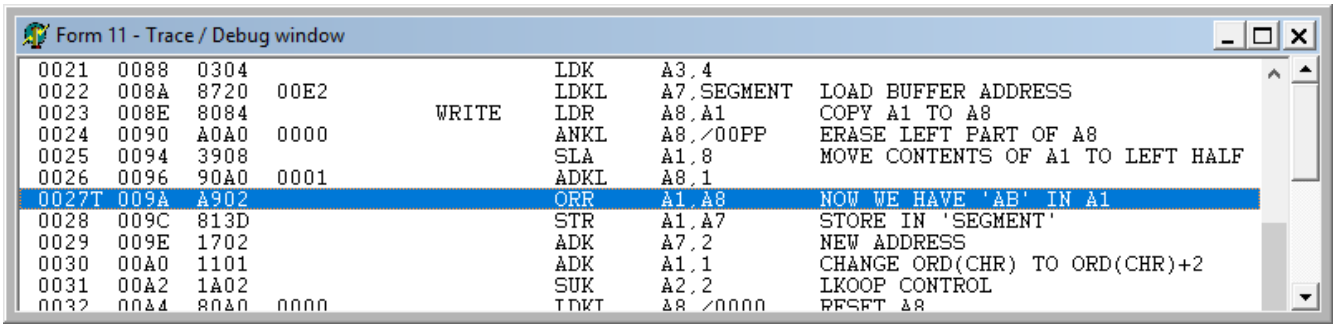
Set trap

Sometimes you have to stop a program at a certain point, e.g. just before the position where you know an error occurs, or where you might need to change a counter, or what migh be the case.

This is done by setting a trap, which prevents the program from executing the instruction. From this point, you can single-step (executing one instruction at a time).

A trap is activated by selecting the line where the trap should be set, and then pressing **Set Trap.**

An example is show below :



Note that a 'T' is written behind the line number.

When you want to remove a specific trap, you must select the line where the trap occurs, and press **Remove trap**

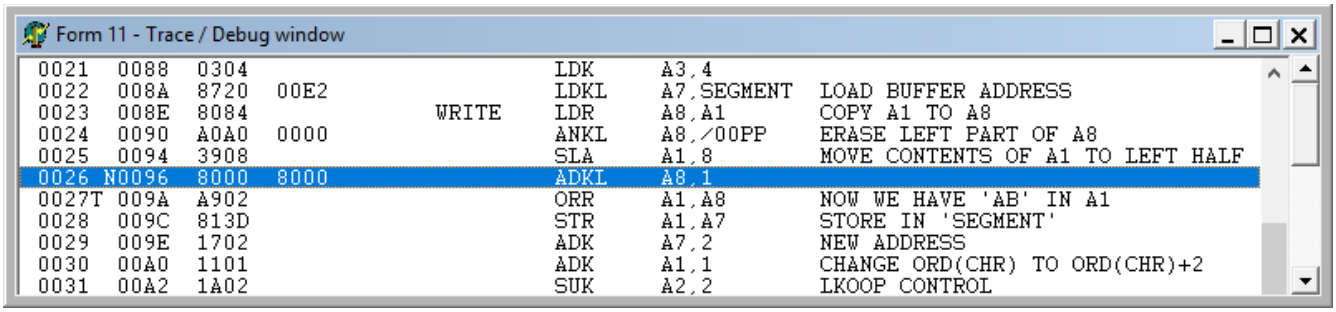
Proceed to next trap is used when you have a number of traps in your program, and you have found out that the instructions between where you are right nowand the next trap, do not need further examination.

Change instruction to NO-OP

The purpose of a NO-OP is to replace an instruction with an instruction that does nothing. This is very useful when you detect that you have an instruction somewhere where it should not be, or when you want to ignore the outcome of an input/output operation (in this case, ignoring faults by overwriting the AB(R) instruction).

Instead of having to make temporary changes and reassemble the program, a NO-OP can be very nice to have.

However, the P800 does not have a defined NO-OP, so instead we use the LR A0,A0 instruction, which corresponds to /8000.



You will note the **N** behind the line number and the 2x /8000. The second instance is written because otherwise the constant would be considered an instruction!

In this case, it would be seen as a LDK R0,1. R0 being a synonym for P, the location counter, you would change the location counter to 1, which would 'modify' your program pretty seriously.

Now we look back at the 'hidden word'.

If we just overwrote the instruction with /8000, we would have lost the original instruction, so what we do, is to hide the original instruction in the 'hidden word'. This enables us to use the

Reestablish Instruction

which overwrites the NO-OP with the original instruction.

Reset all Traps/NO-OPs

removes all traps and no-ops, and reestablishes everything.

The last function in this frame is **JUMP TO SPECIFIC LINE.**

It happens frequently that you find some errors that make it impossible or at least very awkward to continue testing that part of the program.

However, by using **Jump to specific Line** you can go somewhere else, and start testing there. This might save you some work later on.

Hex calculator

The last frame to be discussed, is the Hex calculator.

While developping the Simulator, I frequently had the need for a small calculator to compute offsets, locations, and other hexadecimal based things. Not having a Texas Instruments or other hand-held calculator able to do things in hexadecimal, it was a quick small project to do it myself.

The calculator consists of two parts :

- a real calculator and
- a hex/decimal converter

The calculator expects hexadecimal or decimal values in both fields.

Press **Decimal** or **Hexadecimal**

Enter the values

Select one of the operators

Press =

The lower part of the calculator is a conversion routine.

You can convert values between hex and decimal by writing /... in the uppermost field, and press Convert, or write a decimal value without leading /, which will give you the hex representation of the value.

Appendix A, Using the push buttons in the Simulator

- MC Master Clear Clears memory, interrups, etc.
- RST Read Status Word Displays SW on the 16 lamps
- RR Read Register as specified by the 4 switches
- RM Read memory as specified by the 16 switches
- LR Load Register as specified by the 4 switches
- LM Load Memory
- IPL Initial Program Load Load the lowest 128 bytes with a boot prg
- INST Execute 1 instruction
- RUN Run the program
- INT Interrupt a running program

MC extinguishes all LEDS and resets all tumble switches. It also clears various listboxes, variables and other odds and ends, so you in fact get an 'empty' system.

RST displays the statusword, which again reflects the most recent status of an I/O operation. The meaning of the separate bits can be found in appendix B.

Do not confuse it with **CR** as this is the condition register, set as the result of a compare, an arithmetic operation, or an input-output operation. The Condition Register is explained in Appendix C.

Suppose that we want to see the contents of address /0080. As we remember from one of the examples, the contents are supposed to be /FFFF, /0000

Set the address to /0080 by clicking on tumble switch 9, then press **RM.** This will light up all LEDS, indicating that the value found at /0080 indeed is /FFFF.

Pressing **RM** again, reveals the next word, which /0000. Pressing again, gives /0141, which (not very surprising) is the first instruction.

Memory can be modified by first reading it (**RM**), setting the tumble switches to the address you want inspected, and then press **LM**. It should be noted, that modifying memory without having read it first, must be discouraged. You can check the value through **Inspect/Modify Address**

By single-stepping a few instructions, we can see that A2 is supposed to contain the value /0020 for a space.

Put the value 2 into the 4 tumble switches at the right (OFF, OFF, ON, OFF) and press RR. We should now indeed see /0020.

Putting a new value into R2 is accomplished by setting the value through the switches, and press **LR**

The new value should now be reflected in the **Modify Standard Variables** frame.

Appendix B, Status word

Not all bits are relevant for all devices, and especiall not in connection with the Simulator.

We can hardly expect e.g. a Throughput error, saying that the device is too slow, when we are simulating a Teletype, as the emulated Teletype is shown as a window.

The numbers refer to the number between the LED and the tumble switch.

When the 'Occurs when' is blank, it indicates that the error should not occur.

Bit	Meaning	Occurs when
16	Not Operable	device is switched of or defective
15	Throuput Error	not expected to occur
14	Parity Error / Data Fault	not expected to occur
13	Incorrect length	Wrong length when simulating floppy / disk
12	Program error	not yet relevant
11	End of tape	Emulated cassette is full
7	Seek error	Invalid value in seek command
	BOT / Load point	Occurs after a Rewind command
6	Record not found	Invalid value in seek command
3	Key not found	Invalid key in seek command

Appendix C, Condition Notation

All programs contain arithmetic actions, comparisons and input/output operations, which necessitate the possibility to do enquiries on the result.

For arithmetic it could take the form IF WAGES GREATER 0 THEN...,; etc.

I/O operations need to check for availability of the unit in question, etc.

In the P800, these enquiries are solved by checking the Condition Register, which is done by Compare instruction for arithmetic functions, and Branch instructions for compares and I/O.

You can either test the Condition Register directly with a numeric value, like

```
CWK     A2,0
AB(1)   WRITE
```

where you compare A2 with a constant 0, and make an absolute Branch to WRITE if A2 is greater, but it is very much easier to read to write it as

```
AB(G)   WRITE
```

The following mnemonics are available ;

CR contents	General	Arithmetic	Compare	I/O
0	(0)	(Z)ero	(E)qual	(A)ccepted
1	(1)	(P)os.	(G)reater	(R)efused
2	(2)	(N)eg.	(L)ess	--
3	(3)	(O)verfl.	..	(U)nknown
not 0	(4)	(NZ)	(NE)	(NA)
not 1	(5)	(NP)	(NG)	(NR)
not 2	(6)	(NN)	(NL)	
--	(7)	- - - - - uncoditional - - - - - - - -		

Unconditional branches can be written as

```
AB(7)   <address>     or just   AB   <address>
```

It can also be very useful to have multiple tests immediately after each other, for example when you after an arithmetic operation want to be able to go three different ways depending on the outcome, like

```
                CWK     A2,0
                AB(Z)   RUT1        BRANCH WHEN ZERO
                AB(P)   RUT2        RESULT WAS POSITIVE
                AB(N)   RUT3        RESULT WAS NEGATIVE
ERROR           EQU     *           OVERFLOW HAS OCCURRED
```

--- error handling ---

Appendix D, Instruction formats

The layout of instruction formats is a bit weird, probably caused by the limited memory size of mini computers from that aera.

We have 2 basic formats : aptly called Format 0 and Format 1.

Format 0 : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Bit 0, indicates the basic format : 0 for format 0.

Bits 1..4 define the instruction, implying that we can have only 15 instructions.

Bits 5..7 are used for 2 purposes : either a register 0..7[1], or a condition.

Bits 8..15 serves multiple purposes : it can be a constant, a paramer, or it can be used as an instruction modifier.

An example : an uncoditional branch to address 160 is assembled to /0F0A, in binary 0000 1111 0000 1010, or, split up according to the format shown above :

0 0001 111 00001010

Format 1 : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Bit 0 again indicates the format : a 1 for format 1.

Bits 1..4 indicate the instruction.

Bits 5..7 can be used for a condition, while

Bits 5..8 define the receiving register

Bits 9..10 are a 'modifier'

Bits 11..14 are a register and

Bit 15 is called l/s (least significant), which also an be used as a modifier.

I will give a few examples, to illustrate how confusing an instruction can be.

CWR A3,A5

This instruction compares the contents of A3 and A5, and sets the condition register accordingly. I

Bit 0 will contain a 1, as it is a Format 1 instruction.

Bit 1..4 contain 1101 which is the basic value for a CWR insgtruction,

1 Note that you cannot use all 15 registers in a Format 0 instruction. If you want to use registers 8..15 with a constant, you must use LDKL. The difference being that the constant used in LDKL occupies a word 'behind' the instruction, instead of being a part of the instruction.

Bit 5..8 contain 0011, which is representation for A3,

Bit 9..10 contain the modifier, here 00,

Bit 11..14 contain 0101, which is A5

Bit 15 is 0.

Now then, what is a modifier?

Looking at the syntax for CWR, we see that there are in fact two versions of the instruction. In the manual these are written as follows :

(r1) ↔ (r2) → CR resp.

(r1) ↔ ((r2)) → CR

meaning : compare the CONTENTS of r1 with the CONTENTS of r2 and put the result into the condition register.

The other one means : compare the CONTENTS of r1 with the CONTENTS of of the address R2 is pointing at.

And here the Modifier appears : 00 is the 'straight' compare, and 01 indicates an indexed compare. The index function is activated by not writing CWR as an opcode, but by writing CWR*

Occupying 2 bits, the modifier CAN have 4 values, corresponding to the 4 addressing modes : 'straight', Indexed, Indirect and Indexed Indirect.

The best way to explore the difference, is to peruse the P800 Assembler Manual. Reserve a weekend for this task : it is not for the faint hearted.

A final word is to be said about register use :

Don't use A0 for any purpose, it is the Location counter. In fact, if you use A0 in an instruction, it will be flagged as a fault.

Don't use A15 : it is the stack pointer used in some instructions. Here too, use of A15 will (normally) be flagged